The Report Committee for Zijiang Yang
Certifies that this is the approved version of the following report:

An Experimental Evaluation and Possible Extensions of SyPet

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

Sarfraz Khurshid

Isil Dillig

An Experimental Evaluation and Possible Extensions of SyPet

by

Zijiang Yang, BE

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2017

Acknowledgements

This work was funded in part by the National Science Foundation (NSF Grant Nos. CCF-1319688 and CNS-1239498).

I would first like to thank my report advisor Dr. Sarfraz Khurshid of the Department of Electrical and Computer Engineering at the University of Texas at Austin. The door to Prof. Khurshid office was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this report to be my own work, but steered me in the right the direction whenever he thought I needed it.

I would also like to thank the experts who gave me precious advices during my research: Jinru Hua and Kaiyuan Wang. Without their passionate participation and input, my research could not have been successfully conducted.

I would also like to acknowledge Dr. Isil Dillig of the Department of Computer Science at the University of Texas at Austin as the second reader of this report, and I am gratefully indebted to her for her very valuable comments on this report.

Finally, I must express my very profound gratitude to my parents for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this report. This accomplishment would not have been possible without them. Thank you.

Abstract

An Experimental Evaluation and Possible Extensions of SyPet

Zijiang Yang, MSE

The University of Texas at Austin, 2017

Supervisor: Sarfraz Khurshid

Program synthesis is an automated programming technique that automatically constructs a program which satisfies given specifications. SyPet is a recently published novel component-based synthesis tool that assembles a straight-line Java method body that invokes a sequence of methods from a given set of libraries to implement desired functionality that is defined by a given test suite. In this report, we experimentally evaluate the correctness and performance of the publicly available SyPet implementation, at the black-box level, focusing on the size of test suites. We then demonstrate how SyPet can be extended to support some other applications, such as synthesizing non-straight-line methods and program repair. Finally, we conjecture an alternative technique that is conceptually simpler for synthesizing straight-line methods and present a few initial experimental results.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

The problem of component-based synthesis [1, 2, 3] is to automatically construct a program that is composed from a set of base components. Each base component can be a small piece of program such as a method from an API. This technology can reduce the manual effort of the programmer to read documentations and thus can increase the programmer productivity.

SyPet [3] is a recently published novel tool that can synthesize straight-line Java methods based on a given set of methods. The advantages of SyPet over other component-based synthesis tools are that SyPet can work on a large set of components, e.g. a real-world library, without the need of logical specifications of underlying components. The only information required by SyPet is the expected method signature, a set of tests, and some libraries. With the information gathered, SyPet can automatically generate the implementation details of target method and guarantee that it passes given tests.

The key technology behind SyPet is type-based pruning using a special Petri net. SyPet uses a Petri net to represent the relationships between different methods in a given API, where a reachable path in the Petri net from the initial marking to target marking represents a sequence of method invocations. For each candidate method sequence, SyPet solves a sketch-completion [4] problem using SAT solver, and runs the tests against that candidate to validate it.

Although SyPet is designed to synthesize straight-line Java methods, our experiments demonstrate that we can actually use this tool to synthesize non-straight-line Java methods, specifically ones with if-else branches, as long as the tests are partitioned based on execution paths. Our basic idea is that we can decompose a non-straight-line Java

method synthesis problem into several straight-line Java method synthesis problems, which can be solved by SyPet. Since the test-partitioning phase can be transferred into a set-partitioning problem, we can synthesize certain type of non-straight-line methods in an automated manner.

In addition to demonstrating how to add support for non-straight-line methods, we also explore the use of SyPet for program repair [8, 9, 10]. We show that SyPet can be used to fix errors in a program by replacing a wrong method invocation with a correct one. If we know that a program invokes a wrong method *m* at certain point, we can replace *m* with a new correct method *n* that has the same parameter list and return type as *m*. If we have the scope of candidate methods, we can reduce the program repair problem into a component-based synthesis problem, where *n* is the target method and candidate methods are individual components. That synthesis problem can then be solved by SyPet, and the solution will be a fix to the error in original program. In our experiment, we have successfully fixed a bug in Google Closure Library with the help of SyPet.

Finally, we hypothesize an alternative approach which also synthesizes straight-line Java method using a given set of components. Instead of using Petri nets and SAT solving, our approach uses nondeterministic programming [5] to exhaustively try each possible method sequence and prune them in an execution-driven manner [6]. We use re-execution [7] to try different nondeterministic choices, and invoke the method sequence using Java reflection mechanism, so we do not need to recompile the program each time we try a new candidate. We will outline this simple idea and the initial experimental results later in this report.

The rest of this report is organized as follows: First, we start by experimentally evaluating the correctness and performance of the publicly available implementation of the

SyPet tool (Chapter 2). Then, we demonstrate how to extend SyPet to other applications like non-straight line synthesis and program repair (Chapter 3). Finally, we demonstrate an alternative approach for straight-line method synthesis (Chapter 4).

Chapter 2: Evaluation of SyPet Implementation

We use the publicly available SyPet implementation (Version 1.0) in our experimental evaluation. We specifically investigate the following research question: how does SyPet correctness and performance depend on the number of given tests? In this chapter, we will describe our experiments and report our findings.

I. Experimental Benchmarks

In order to figure out how does SyPet work under different number of tests, we expand their original benchmarks used in the POPL 2017 paper [3] by adding more tests and run different number of tests respectively for each original benchmark. To make our evaluation more thorough, we add 2 additional benchmarks for each library presented in the original paper, and we also add some new benchmarks based on our self-written libraries. We have 47 benchmarks and 376 tests in total. The statistics of our new benchmarks are shown in Table 1.

Library	Number of benchmarks	Number of tests for each benchmark
Apache math	11	8
Geometry	8	8
Joda time	9	8
Jsoup, dom, text	10	8
Self-written library	9	8

Table 1: Statistics of SyPet benchmarks

II. Results and Analysis

For each benchmark, we run SyPet 8 times, with 1 to 8 tests respectively. So, we run SyPet 376 times in total. We collect the information about synthesis time, number of Petri net paths we have explored, number of program candidates we have tried, number of components in the solution, number of holes in the solution, and if the solution is correct. All our experiments are performed in a Macbook Pro with 2.7 GHz Intel core i5 processor and 8 GB memory.

The detailed results can be found in appendix of this report. From the results, we can get 3 conclusions regarding the number of test cases:

1. *Number of test cases affects correctness. More test cases are likely to result in better solutions.*
2. *Number of test cases does not affect performance much if the result is correct. But if the result is incorrect, having more test cases may result in longer execution time.*
3. *We need 1.26 tests to get correct solution on average.*

The first conclusion is easy to understand. SyPet always guarantees that the result passes all given test cases. But if the given test cases are not enough to identify the right implementation, it may result in wrong solution. Suppose we want to synthesize a method which calculates the absolute value of a double input using *java.lang* library. If we use tests in Figure 1, we may result in getting a wrong implementation *java.lang.Math.sqrt(double)* instead of *java.lang.Math.abs(double)*. That is because both *abs* and *sqrt* return 0 when input is 0, so both methods pass given tests. However, if we have more tests like Figure 2, we will be able to filter out the wrong method and possibly get the correct solution. That

demostrates having more tests will give a more precise description of target method and results in better solutions.

```
boolean test0() {
    return abs(0) == 0;
}
```

Figure 1: Example of inadequate tests for *int abs(int)*

```
boolean test0() {
    return abs(0) == 0;
}

boolean test1() {
    return abs(1) == 1;
}

boolean test2() {
    return abs(-1) == 1;
}
```

Figure 2: Example of good tests for *int abs(int)*

The second conclusion can be understood in the same way. Having fewer tests makes the specification more general and thus results in more accepted solutions. As a result, it takes less time to find the first acceptable solution. With number of tests increasing, the specification is becoming more and more detailed, and the execution time is becoming longer and longer. Once the correct solution is found, it always passes newer tests, so the search process always stops at that solution. As a result, the execution time stops increasing.

These conclusions show that we do not need too many redundant tests. Instead, we only need a small number of tests that can precisely describe the behavior of target method.

In this way, we not only save test execution time, but also save programmer's time for test writing.

Chapter 3: New Applications of SyPet

The original SyPet tool is designed to synthesize straight-line Java methods. In this chapter, we demonstrate two more applications that can be reduced to straight-line method synthesis problems, which then can be solved by SyPet.

I. Component-based Synthesis for Non-straight-line Methods

While SyPet is an efficient tool in synthesizing straight-line Java methods, a lot of real-life methods are not straight-line and therefore cannot be synthesized by SyPet. However, our work shows that we can use SyPet as an enabling technology to synthesize such methods.

1. Test-partition Based Approach for Single If-Else Statement

Let's consider the simplest case, where the target method contains a single if-else statement. The basic idea is to reduce such a synthesis problem to several straight-line method synthesis problems. Because we know that target method contains two execution paths, we can treat each individual execution path as a straight-line method, whose input arguments and return type are the same as those of the original method. In this way, we can reduce the original synthesis problem to several sub-problems that can be solved by SyPet.

Suppose we want to synthesize the method in Figure 3. The original method in Figure 3 can be abstracted as a new method whose if-body and else-body are replaced by a single method call. The control flow graph of the abstraction in Figure 3 can be found in Figure 4. From the control flow graph, we notice that *ifbody()* and *elsebody()* cannot be executed in a single method invocation. In other words, if *ifbody()* is executed, *elsebody()* cannot be executed during the same method call. Due to this property, it is possible to write

test cases that only execute *ifbody()* or *elsebody()*. If we have these test cases, we can synthesize *ifbody()* and *elsebody()* respectively, by applying test cases that only execute *ifbody()* or *elsebody()*.

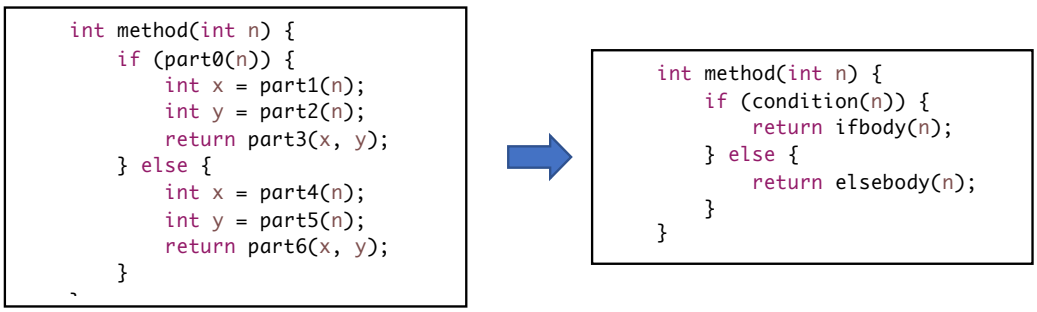


Figure 3: A non-straight-line method and its abstraction

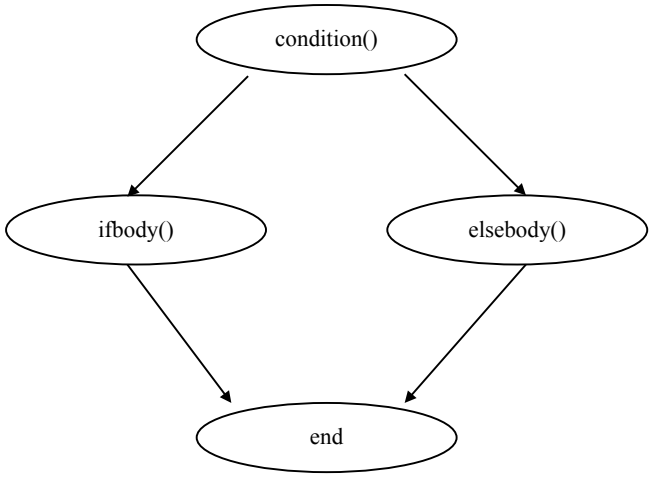


Figure 4: Control flow graph of abstracted method in Figure 3

If we already have the implementations of *ifbody()* and *elsebody()*, the last step we need to do is synthesizing the *condition()*. Since *condition()* is the only unknown component, we can put all test cases together and let SyPet figure out the right implementation of *condition()* based on the *ifbody()* and *elsebody()* we have synthesized in previous steps.

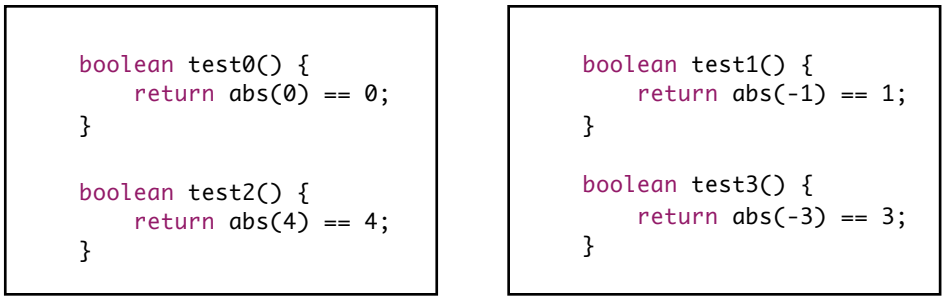
The algorithm to synthesize a single if-else statement is called Algorithm 1, whose inputs are method signature, library, and partitioned test cases. The details of Algorithm 1 can be found in Figure 5.

```
synthesize_single_if_statement:
input:  signature: String, library: String,
        tests1: JavaFile, tests2: JavaFile
output: synthesized method or failure

ifbody :=  SYPET(signature, library, tests1)
elsebody := SYPET (signature, library, tests2)
skeleton :=  COMBINE_IF_ELSE_BODY(ifbody, elsebody)
tests3 := UNION(tests1, tests2, skeleton)
condition := SYPET("booean condition(args)", library, tests3)
result := COMBINE_METHOD(ifbody, elsebody, condition)

return result
```

Figure 5: Algorithm 1, an algorithm to synthesize single if-else statement

Here is an example. If we want to synthesize a method *int abs(int)* which returns the absolute value of its input, we can apply Algorithm 1 based on test cases shown in Figure 6. Firstly, we need to run SyPet against tests in partition 1 and get the first solution, which will be an implementation that simply returns the argument. Then, we run SyPet

against tests in partition 2 and get the second solution, which will be an implementation that negates the input argument. We know that solution 1 and solution 2 are different branches of the actual implementation of *int abs(int)*, so we can write the skeleton of *int abs(int)* like Figure 7.



```
boolean test0() {
    return abs(0) == 0;
}

boolean test2() {
    return abs(4) == 4;
}
```

```
boolean test1() {
    return abs(-1) == 1;
}

boolean test3() {
    return abs(-3) == 3;
}
```

Partition 1 Partition 2

Figure 6: Partition of tests for method *int abs(int)*

```
int abs(int n) {
    if(condition(n)) {
        int a = mehtod_return_argument(n);
        return a;
    } else {
        int b = method_negate_argument(n);
        return b;
    }
}
```

Figure 7: The skeleton of method *int abs(int)*

Then, we can put all these tests together, use the skeleton as a helper method, and let SyPet synthesize the method *boolean condition(int)*. Once SyPet gives an accepted

solution, we can put the solution into the skeleton and get a final solution to the original method *int abs(int)*.

2. Test-partition Based Approach on Chaining If-Else Statements

In this part, we extend the previous algorithm to make it applicable to chaining if-else statements. An if-else chain is a sequence of if-else statements where an else-body overlaps the next if-body. Figure 8 is an example of an if-else chain inside a method.

```
int method(int n) {
    if (condition0(n)) {
        return body0(n);
    } else if (condition1(n)) {
        return body1(n);
    } else if (condition2(n)) {
        return body2(n);
    } else {
        return body3(n);
    }
}
```

Figure 8: An abstract method with an if-else chain

Suppose we want to synthesize the method in Figure 8, we can use an approach similar to Algorithm 1. Firstly, we need to partition the test cases based on execution paths. In this example, we need 4 partitions. Then, we synthesize each if-body or else-body based on its partitioned tests. Finally, we synthesize the conditions in bottom-up order. Specifically, we first put partition 2, 3 together and synthesize *condition2()*. Then, we put partition 1, 2, 3 together and synthesize *condition1()*. Finally, we put all partitions together and synthesize *condition0()*. We call this approach Algorithm 2, which is shown in Figure 9.

```
synthesize_chaining_if_statements:
input:  signature: String, library: String,
          tests: JavaFile[n]
output: synthesized method or failure

Method[n] bodies
for i from 0 until n-1, do:
      bodies[i] := SYPET(signature, library, tests[i])
result := bodies[n-1]
for i from  n-2 until 0, do:
      ifbody := bodies[i]
      elsebody := result
      skeleton := COMBINE_IF_ELSE_BODY(ifbody, elsebody)
      test := UNION(tests[i .. n-1], skeleton)
      condition := SYPET("boolean condition(args)", library, test)
      result := COMBINE_METHOD(ifbody, elsebody, condition)

return result
```

Figure 9: Algorithm 2, an algorithm to synthesize chaining if-else statements

In our experiments, we have successfully used Algorithm 2 to synthesize a method that categorizes a triangle into one of three classes: acute triangle, obtuse triangle, and right triangle. Our library is a self-written library which contains methods that return category labels and methods that distinguish different kinds of triangles. We have run SyPet 5 times to get the final solution, which is an if-else chain with 2 conditions and 3 bodies. Each execution takes 4.5 seconds on average.

3. Automated Synthesis of Chaining If-Else Statements

Although Algorithm 2 works well in synthesizing chaining if-else statements, it requires the programmer to manually partition the test cases based on execution paths. Sometimes the problem is not as intuitive as those in our experiments, in which case

programmer finds it hard to partition test cases. In this part, we describe another algorithm that automatically synthesize chaining if-else statements without the needs of partitioning test cases manually.

Our new algorithm has two phases: test partitioning phase and synthesis phase. The test partitioning phase is to automatically partition test cases into a few subsets. This is the same as a traditional set-partitioning problem, except that we disallow empty sets in our problem. When test partitioning is done, we can synthesize all the if-else bodies based on the partitions. After that, we can synthesize if-conditions, by trying all possible orders of if-else bodies. That is because different order of if-else bodies can result in completely different conditions. An example is shown in Figure 10. Even though these 2 methods have same if-else bodies, their different orders result in completely different conditions. If our library only has *isRight()* method, the second order will make SyPet fail to find a solution.

```
String categorizeTriangle(Triangle t) {
    if(isAcute(t)) {
        return "acute";
    } else if(isRight(t)) {
        return "right";
    } else {
        return "obtuse";
    }
}
```

```
String categorizeTriangle(Triangle t) {
    if(isAcute(t)) {
        return "acute";
    } else if(isObtuse(t)) {
        return "obtuse";
    } else {
        return "right";
    }
}
```

Figure 10: Different order of if-else bodies results in different conditions

Our algorithm to automatically synthesize chaining if-else statements is called Algorithm 3, which is shown in Figure 11.

```
auto_synthesize_chaining_if_statements:
input: signature: String, library: String,
        tests: JavaFile, num_branches: int
output: synthesized method or failure

for each partitioned_tests in PARTITION_TESTS(tests, num_branches), do:
    Method[num_branches] bodies
    for i from 0 until num_branches-1, do:
        bodies[i] := SYPET(signature, library, partitioned_tests[i])
    for each (tests_permutation, bodies_permutation) in
PERMUTATIONS(partitioned_tests, bodies), do:
        conditions = SYNTHESIZE_ALL_CONDITIONS(library, tests_permutation,
bodies_permutation)
        result = COMBINE_METHOD(bodies_permutation, conditions)
        if NO ERROR :
            return result
```

Figure 11: Algorithm 3, an algorithm to automatically synthesize chaining if-else statements

We have successfully used Algorithm 3 to synthesize a method *int abs(int)* that contains a single if-else statement. We have tried different size of test cases from 2 to 5. The results can be found in Table 2. Our test cases are shown in Figure 12. Figure 13 is a screenshot of the output of our Algorithm.

Number of tests	Number of partitions tried	Result partition	Total time (seconds)
2	1	{0}, {1}	13
3	2	{0, 2}, {1}	758
4	2	{0, 2}, {1, 3}	812
5	7	{0, 2, 4}, {1, 3}	4514

Table 2: Experimental results of Algorithm 3

```java
public static boolean test0() throws Throwable {
    return abs(0) == 0;
}

public static boolean test1() throws Throwable {
    return abs(-Integer.MAX_VALUE) == Integer.MAX_VALUE;
}

public static boolean test2() throws Throwable {
    return abs(5) == 5;
}

public static boolean test3() throws Throwable {
    return abs(-100) == 100;
}

public static boolean test4() throws Throwable {
    return abs(4) == 4;
}
```

Figure 12: Test cases used in experiments of Algorithm 3

```
Solution:
-----------------------------------------------------
public static int abs(int val) throws Throwable {
    if(condition(val)) {
        return if_body(val);
    } else {
        return else_body(val);
    }
}

public static int if_body(int val) throws Throwable {
    int sypet_var2 = library.Library.forward(val);
    return sypet_var2;
}

public static int else_body(int val) throws Throwable {
    int sypet_var4 = library.Library.negate(val);
    return sypet_var4;
}

public static boolean condition(int val) throws Throwable {
    boolean sypet_var2 = library.Library.isPositive(val);
    return sypet_var2;
}
-----------------------------------------------------
Script running time: 4514 seconds. Combinations tried: 7.
```

Figure 13: Screenshot of experimental output of Algorithm 3

4. Limitations of Test-partition Based Approach

We have demonstrated that SyPet can be used to synthesize non-straight-line methods with if-else statements. However, not all methods with if-else statements can be synthesized by SyPet. We have generalized that a method must meet two requirements in order to be synthesized with test-partition based approach:

1. *The method can be rewritten into an equivalent form with single if-else chain and no statements outside of if-else clauses.*
2. *The conditions do not change program state.*

The first requirement is obvious because our solutions are always in a form where everything is inside a single if-else chain. If a method is not in such a form, there is no way that SyPet can get the correct solution.

The second requirement aims to ensure the complete separation between conditions and if-else bodies. That is because our test-partition based approach synthesizes if-else bodies without considering conditions. If the conditions cannot be separated from if-else bodies, the result may be incorrect.

```
int method(Iterator<Integer> it) {
    if(it.next().equals(0)) {
        return it.next();
    } else {
        return sum(it.next(), it.next());
    }
}
```

Figure 14: An example of if-condition that changes program state

Figure 14 shows such an example. If the first element in iterator is 0, the method will return the second element. Otherwise, it will return the sum of second and third elements. According to this logic, the synthesized if-body and else-body will be Figure 15, which are incorrect because they redundantly include the condition.

```
int ifbody(Iterator<Integer> it) {
    it.next();
    return it.next();
}

int elsebody(Iterator<Integer> it) {
    it.next();
    return sum(it.next(), it.next());
}
```

Figure 15: Synthesized if-body and else-body of method in Figure 14

Due to above reasons, both requirements must be satisfied by a method in order to be synthesized with our test-partition based algorithms.

II. Program Repair

Program repair is another important research area in Software Engineering [8, 9, 10]. It requires a program to automatically repair software bugs without the intervention of a human programmer. At a high level, program repair techniques generate a patch for a program in two steps. The first step is patch generation, which analyzes the original program and produces one or more candidate patches. The second step is patch validation, which validates the produced candidate patches from the previous step with either a formal specification or a test suite of the program.

There are key similarities between the procedure of component-based synthesis and the procedure of program repair. They both generate candidates and validate them with

some given specifications, e.g. test suites. From a high-level intuition, it is possible to reduce a program repair problem to a component-based synthesis problem.

Our research shows we can use SyPet to solve such problems, as long as we know the location of the bug and the repairing patch is a sequence of method invocations. We can replace that method sequence with a "hole", and let SyPet figure out the implementation of that "hole" using existing methods. If an implementation makes the program pass given test suites, we know this implementation is a correct fix to the bug. In this way, a program repair problem is reduced to a component-based synthesis problem which can be solved by SyPet.

Though it is possible to use SyPet for program repair problems, it is still not an easy task because SyPet does not provide such interfaces. The first problem is how to let SyPet insert candidate code into the "hole" in buggy program. That is not supported by current SyPet implementation because candidate method sequences are only accessible in test files and are not exposed to users. Another problem is how to transfer the original test suites into SyPet-compatible tests. SyPet requires a boolean test method instead of JUnit test suites, so we need some mechanisms to transfer JUnit test suites into a single test method.

We use following techniques to solve above problems. Assume the original program invokes a wrong method *w* at some place in class *c*. To fix the bug, we need to do following steps:

1. Define a new interface *i*, and declare a method *m* inside *i*. The signature of *m* is the same as the signature of *w*.
2. Add a new public static field *f* in class *c*. The type of *f* is *i*.
3. At the buggy location, replace the invocation of *w* to *f.m*.

4. Compile the whole program, add the compiled classes as a library for SyPet.
5. Write a standard SyPet-styled test file with a single test method.
6. At the beginning of test method, define an anonymous instance *a* which implements *i* and overrides *m* by invoking generated candidate method. Assign *a* to *c.f*.
7. In the test method, invoke original JUnit test suites programmatically, then check the result. If JUnits result is successful, return true. Otherwise, return false.
8. Run SyPet. The solution will be the correct method invocation.

The above approach uses polymorphism to insert candidate methods into the "hole". It firstly replaces the wrong method invocation with an instance method invocation from a static field. Then, it updates that static field before each execution of test suites. In this way, the candidate method generated by SyPet will be invoked during test execution.

We have tested this approach with a real-life bug in defects4j. In our experiments, SyPet successfully fixes the bug 104 in Google Closure Library in 19 seconds. Our test file looks like Figure 16.

```
    public static boolean test0() throws Throwable {
        com.google.javascript.rhino.jstype.UnionType.solution = new
com.google.javascript.rhino.jstype.SypetSolution() {
            @Override
            public boolean solution(com.google.javascript.rhino.jstype.JSType type)  {
                try {
                    return mySolution(type);
                } catch(Throwable e) {
                    throw new RuntimeException(e.toString());
                }
            }
        };
        org.junit.runner.Result res = new
org.junit.runner.JUnitCore().run(com.google.javascript.rhino.jstype.UnionTypeTest.class);
        return res.wasSuccessful();
    }
```

Figure 16: Test file for program repair experiment

Chapter 4: Idea of an Alternative Approach for Component-based Synthesis

In this chapter, we briefly hypothesize an alternative approach which also synthesizes straight-line Java methods using a given set of components in the spirit of SyPet but is conceptually simpler in functionality. Our approach requires the same information as SyPet: a set of candidate methods, the signature of target method, and some tests. Instead of solving Petri net reachability problems and SAT problems like SyPet, our approach uses nondeterministic programming [5] to exhaustively try each possible method sequence and prune them in execution-driven manner [6].

I. Overview

The basic idea of our approach is very simple. If we have a list of possible methods, we can try every sequence exhaustively in a backtracking manner. For each method sequence, we check if it satisfies the specifications. If we find an acceptable method sequence, we stop searching and return the result. This kind of backtracking is based on model checkers. We use a re-execution model checker to choose different methods non-deterministically, and invoke these methods using Java reflection. In this way, we can save time by not recompiling candidate method sequences each time we run test suites.

To prune method sequences, we do not apply complex pruning strategies like Petri net reachability checking. Instead, we apply some very simple pruning strategies, which are described in the next section. Though less effective pruning in our approach may result in more candidate checking, it has the potential to save a lot of time by avoiding complex constraint solving problems.

II. Pruning Strategy

We demonstrate three simple pruning strategies: type-based pruning, variable-based pruning, and execution-driven pruning.

Firstly, we prune methods based on its parameter types and return type. If a method *m* in sequence *S* requires an argument type *T*, but *T* neither appears in parameter list nor is a return type of some previous methods in *S*, we can safely prune *m*. Similarly, if the target method returns type *T*, but no method in sequence *S* has return type *T*, we can safely prune *S*.

Next, we prune method sequences which produce unused intermediate variables. This strategy is same as SyPet, because method sequences that produce unused variables are unlikely to be correct [3].

Finally, we prune method sequences in execution-driven manner. Our execution-driven pruning is very straightforward. We observe each individual method during test executions. If any method throws an exception, we do not need to consider any method sequences with the same prefix. For example, if we have method sequence [*m1, m2, m3, m4*], and *m2* throws an exception during a test execution, we know that any sequences start with *m1, m2* will throw the same exception. That is because every sequence starts with *m1* will share the same state when *m1* have been executed. Starting from that state, *m2* will always perform the same behavior. Thus, we can safely ignore sequences [*m1, m2, …*].

III. Experimental Results

We have built a simple prototype of our approach and performed a few experiments. Table 3 shows the experimental results of our tool. Table 4 shows the results of SyPet for these benchmarks. All these experiments are performed in a Macbook Pro with 2.7 GHz Intel core i5 processor and 8 GB memory.

Benchmark id	Library	Number of tests	Number of candidates explored	Number of methods	Total time (seconds)	Result
8	Apache common math	8	3846	3	4	correct
9	Apache common math	8	850299	4	467	correct
21	Joda time	8	356435	-	289	No solution
22	Joda time	8	624956	4	318	correct

Table 3: Experimental results of our prototype

Benchmark id	Library	Number of tests	Number of candidates explored	Number of methods	Total time (seconds)	Result
8	Apache common math	8	1094	5	241	correct
9	Apache common math	8	-	-	393	Out of Memory Error
21	Joda time	8	4716	4	414	correct
22	Joda time	8	620	4	68	correct

Table 4: Experimental results of SyPet for same benchmarks as shown in Table 3

From the experimental results, we know that both tool fails in one benchmark. Our tool fails in benchmark 21 and SyPet fails in benchmark 9. SyPet can actually complete this benchmark successfully if given more memory but failed on our experimental computer that did not have the required amount of memory. Our tool fails because of current limitations in our approach in handling static members.

Chapter 5: Conclusion

In this report, we have described our black-box experiments on SyPet and presented our findings. Then, we have demonstrated that SyPet can synthesize non-straight-line methods with chaining if-else statements. We have also demonstrated that we can use SyPet to solve program repair problems, as long as the repairing patch is a sequence of method invocations. Finally, we have hypothesized an alternative approach which also synthesizes straight-line Java methods using a given set of components but is conceptually simpler, and presented our experimental results.

Appendix

EXPERIMENT RESULTS OF SYPET

ID	Description	#Tests	Pruning	Synthesis Time (s)	#Paths	#Progs	#Comps	#Holes	Result	Notes
1	Compute the pseudo-inverse of a matrix	1	ON	8.43	256	511	3	4	correct	-
		2	ON	9.07	256	511	3	4	correct	-
		3	ON	9.03	256	511	3	4	correct	-
		4	ON	9.09	256	511	3	4	correct	-
		5	ON	9.41	256	511	3	4	correct	-
		6	ON	9.11	256	511	3	4	correct	-
		7	ON	8.89	256	511	3	4	correct	-
		8	ON	8.31	256	511	3	4	correct	-
2	Compute the inner product between two vectors	1	ON	0.33	1	1	3	5	correct	-
		2	ON	0.26	1	1	3	5	correct	-
		3	ON	0.28	1	1	3	5	correct	-
		4	ON	0.29	1	1	3	5	correct	-
		5	ON	0.26	1	1	3	5	correct	-
		6	ON	0.26	1	1	3	5	correct	-
		7	ON	0.29	1	1	3	5	correct	-
		8	ON	0.29	1	1	3	5	correct	-
3	Determine the roots of a polynomial equation	1	ON	0.96	7	13	3	5	correct	-
		2	ON	0.72	7	13	3	5	correct	-
		3	ON	0.78	7	13	3	5	correct	-
		4	ON	0.82	7	13	3	5	correct	-
		5	ON	0.77	7	13	3	5	correct	-
		6	ON	0.8	7	13	3	5	correct	-
		7	ON	1.17	7	13	3	5	correct	-
		8	ON	0.82	7	13	3	5	correct	-
4	Compute the singular value decomposition of a matrix	1	ON	0.11	1	1	3	4	correct	-
		2	ON	0.11	1	1	3	4	correct	-
		3	ON	0.12	1	1	3	4	correct	-
		4	ON	0.11	1	1	3	4	correct	-
		5	ON	0.11	1	1	3	4	correct	-
		6	ON	0.13	1	1	3	4	correct	-

ID	Description	#Tests	Pruning	Synthesis Time (s)	#Paths	#Progs	#Comps	#Holes	Result	Notes
		7	ON	0.12	1	1	3	4	correct	-
		8	ON	0.12	1	1	3	4	correct	-
5	Invert a square matrix	1	ON	0.65	16	31	3	4	correct	-
		2	ON	0.7	16	31	3	4	correct	-
		3	ON	0.67	16	31	3	4	correct	-
		4	ON	0.67	16	31	3	4	correct	-
		5	ON	0.66	16	31	3	4	correct	-
		6	ON	0.73	16	31	3	4	correct	-
		7	ON	0.76	16	31	3	4	correct	-
		8	ON	0.66	16	31	3	4	correct	-
6	Solve a system of linear equations	1	ON	34	788	1600	6	8	correct	-
		2	ON	33.21	788	1600	6	8	correct	-
		3	ON	32.82	788	1600	6	8	correct	-
		4	ON	34.88	788	1600	6	8	correct	-
		5	ON	35.04	788	1600	6	8	correct	-
		6	ON	32.9	788	1600	6	8	correct	-
		7	ON	34.17	788	1600	6	8	correct	-
		8	ON	32.83	788	1600	6	8	correct	-
7	Compute the outer product between two vectors	1	ON	4.83	14	48	4	6	correct	-
		2	ON	5.06	14	48	4	6	correct	-
		3	ON	5.37	14	48	4	6	correct	-
		4	ON	2.83	14	48	4	6	correct	-
		5	ON	5.15	14	48	4	6	correct	-
		6	ON	2.59	14	48	4	6	correct	-
		7	ON	2.6	14	48	4	6	correct	-
		8	ON	2.77	14	48	4	6	correct	-
8	Predict a value from a sample by linear regression	1	ON	176.29	534	1094	6	6	correct	-
		2	ON	175.53	534	1094	6	6	correct	-
		3	ON	176.67	534	1094	6	6	correct	-
		4	ON	176.4	534	1094	6	6	correct	-
		5	ON	185.44	534	1094	6	6	correct	-
		6	ON	179.46	534	1094	6	6	correct	-
		7	ON	181.81	534	1094	6	6	correct	-

ID	Description	#Tests	Pruning	Synthesis Time (s)	#Paths	#Progs	#Comps	#Holes	Result	Notes
		8	ON	188.79	534	1094	6	6	correct	-
9	Compute the ith eigenvalue of a matrix	1	ON	7.05	117	263	4	6	incorrect	-
		2	OFF	-	-	-	-	-	incorrect	Fails to sythesize when pruning is on; OutOfMemoryError
		3	OFF	-	-	-	-	-	incorrect	Fails to sythesize when pruning is on; OutOfMemoryError
		4	OFF	-	-	-	-	-	incorrect	Fails to sythesize when pruning is on; OutOfMemoryError
		5	OFF	-	-	-	-	-	incorrect	Fails to sythesize when pruning is on; OutOfMemoryError
		6	OFF	-	-	-	-	-	incorrect	Fails to sythesize when pruning is on; OutOfMemoryError
		7	OFF	-	-	-	-	-	incorrect	Fails to sythesize when pruning is on; OutOfMemoryError
		8	OFF	-	-	-	-	-	incorrect	Fails to sythesize when pruning is on; OutOfMemoryError
10	Compute the transpose of a matrix	1	ON	0.05	1	1	1	2	incorrect	-
		2	ON	0.34	8	15	3	4	correct	-
		3	ON	0.33	8	15	3	4	correct	-
		4	ON	0.35	8	15	3	4	correct	-
		5	ON	0.53	8	15	3	4	correct	-
		6	ON	0.35	8	15	3	4	correct	-
		7	ON	0.37	8	15	3	4	correct	-
		8	ON	0.34	8	15	3	4	correct	-
11	Compute the sum of two matrices	1	ON	0.64	5	17	4	6	correct	-
		2	ON	0.52	5	17	4	6	correct	-
		3	ON	0.54	5	17	4	6	correct	-
		4	ON	0.52	5	17	4	6	correct	-
		5	ON	0.59	5	17	4	6	correct	-
		6	ON	0.53	5	17	4	6	correct	-
		7	ON	0.53	5	17	4	6	correct	-
		8	ON	0.57	5	17	4	6	correct	-
12	Scale a rectangle by a given ratio	1	ON	1.92	78	275	4	7	correct	-
		2	ON	1.97	78	275	4	7	correct	-
		3	ON	1.53	78	275	4	7	correct	-
		4	ON	2	78	275	4	7	correct	-

ID	Description	#Tests	Pruning	Synthesis Time (s)	#Paths	#Progs	#Comps	#Holes	Result	Notes
		5	ON	1.77	78	275	4	7	correct	-
		6	ON	1.66	78	275	4	7	correct	-
		7	ON	1.82	78	275	4	7	correct	-
		8	ON	1.69	78	275	4	7	correct	-
13	Shear a rectangle and get its tight rectangular bounds	1	ON	2.1	79	282	4	7	correct	-
		2	ON	2.19	79	282	4	7	correct	-
		3	ON	2.42	79	282	4	7	correct	-
		4	ON	2.31	79	282	4	7	correct	-
		5	ON	2.45	79	282	4	7	correct	-
		6	ON	2.08	79	282	4	7	correct	-
		7	ON	2.13	79	282	4	7	correct	-
		8	ON	2.37	79	282	4	7	correct	-
14	Rotate a rectangle about the origin by the specified number of quadrants	1	ON	0.43	9	21	4	6	correct	-
		2	ON	0.59	9	21	4	6	correct	-
		3	ON	0.43	9	21	4	6	correct	-
		4	ON	0.48	9	21	4	6	correct	-
		5	ON	0.42	9	21	4	6	correct	-
		6	ON	0.46	9	21	4	6	correct	-
		7	ON	0.56	9	21	4	6	correct	-
		8	ON	0.5	9	21	4	6	correct	-
15	Rotate two dimensional geometry object by the specified angle about a point	1	ON	0.1	1	1	3	6	incorrect	-
		2	ON	2.97	67	225	5	8	correct	-
		3	ON	3.2	67	225	5	8	correct	-
		4	ON	2.72	67	225	5	8	correct	-
		5	ON	2.91	67	225	5	8	correct	-
		6	ON	3.08	67	225	5	8	correct	-
		7	ON	2.75	67	225	5	8	correct	-
		8	ON	2.98	67	225	5	8	correct	-
16	Perform a translation on a given rectangle	1	ON	0.92	41	156	4	7	correct	-
		2	ON	1	41	156	4	7	correct	-
		3	ON	1.12	41	156	4	7	correct	-
		4	ON	1.1	41	156	4	7	correct	-
		5	ON	1.15	41	156	4	7	correct	-

ID	Description	#Tests	Pruning	Synthesis Time (s)	#Paths	#Progs	#Comps	#Holes	Result	Notes
		6	ON	1.1	41	156	4	7	correct	-
		7	ON	1.02	41	156	4	7	correct	-
		8	ON	1.16	41	156	4	7	correct	-
17	Compute the intersection of a rectangle and the rectangular bounds of an ellipse	1	ON	0.05	1	1	3	5	correct	-
		2	ON	0.06	1	1	3	5	correct	-
		3	ON	0.06	1	1	3	5	correct	-
		4	ON	0.06	1	1	3	5	correct	-
		5	ON	0.05	1	1	3	5	correct	-
		6	ON	0.07	1	1	3	5	correct	-
		7	ON	0.07	1	1	3	5	correct	-
		8	ON	0.05	1	1	3	5	correct	-
18	Check if a point is inside a rectangle	1	ON	0.08	1	1	3	5	incorrect	-
		2	ON	17.45	423	890	6	8	correct	-
		3	ON	18.65	423	890	6	8	correct	-
		4	ON	19.3	423	890	6	8	correct	-
		5	ON	18.3	423	890	6	8	correct	-
		6	ON	18.49	423	890	6	8	correct	-
		7	ON	18.04	423	890	6	8	correct	-
		8	ON	17.83	423	890	6	8	correct	-
19	Check if a line segment intersects a rectangle.	1	ON	0.16	1	1	3	5	incorrect	-
		2	ON	0.55	6	16	4	7	incorrect	-
		3	ON	0.61	6	16	4	7	incorrect	-
		4	ON	0.49	6	16	4	7	incorrect	-
		5	ON	0.51	6	16	4	7	incorrect	-
		6	ON	0.53	6	16	4	7	incorrect	-
		7	ON	0.51	6	16	4	7	incorrect	-
		8	OFF	-	-	-	-	-	incorrect	Fails to sythesize when pruning is on; OutOfMemoryError
20	Compute number of days since the specified date	1	ON	6.43	78	156	3	4	correct	-
		2	ON	6.36	78	156	3	4	correct	-
		3	ON	6.45	78	156	3	4	correct	-
		4	ON	6.49	78	156	3	4	correct	-
		5	ON	6.25	78	156	3	4	correct	-

ID	Description	#Tests	Pruning	Synthesis Time (s)	#Paths	#Progs	#Comps	#Holes	Result	Notes
		6	ON	6.4	78	156	3	4	correct	-
		7	ON	6.36	78	156	3	4	correct	-
		8	ON	6.33	78	156	3	4	correct	-
21	Compute the number of days between two dates considering timezone	1	ON	0.15	1	2	2	4	incorrect	-
		2	ON	168.22	769	4716	4	6	correct	-
		3	ON	167.88	769	4716	4	6	correct	-
		4	ON	168.42	769	4716	4	6	correct	-
		5	ON	166.8	769	4716	4	6	correct	-
		6	ON	168.89	769	4716	4	6	correct	-
		7	ON	167.73	769	4716	4	6	correct	-
		8	ON	168.56	769	4716	4	6	correct	-
22	Determine if a given year is a leap year	1	ON	0.39	2	3	2	3	incorrect	-
		2	ON	38.42	308	620	4	5	correct	-
		3	ON	38.36	308	620	4	5	correct	-
		4	ON	37.67	308	620	4	5	correct	-
		5	ON	38.24	308	620	4	5	correct	-
		6	ON	38.35	308	620	4	5	correct	-
		7	ON	37.19	308	620	4	5	correct	-
		8	ON	37.27	308	620	4	5	correct	-
23	Return the day of a date string	1	ON	0.81	1	1	3	5	correct	-
		2	ON	0.95	1	1	3	5	correct	-
		3	ON	0.78	1	1	3	5	correct	-
		4	ON	0.65	1	1	3	5	correct	-
		5	ON	0.8	1	1	3	5	correct	-
		6	ON	0.69	1	1	3	5	correct	-
		7	ON	0.76	1	1	3	5	correct	-
		8	ON	0.72	1	1	3	5	correct	-
24	Find the number of days of a month in a date string	1	ON	54.52	147	441	4	6	correct	-
		2	ON	51.18	147	441	4	6	correct	-
		3	ON	50.23	147	441	4	6	correct	-
		4	ON	50.16	147	441	4	6	correct	-
		5	ON	49.57	147	441	4	6	correct	-
		6	ON	51.25	147	441	4	6	correct	-

ID	Description	#Tests	Pruning	Synthesis Time (s)	#Paths	#Progs	#Comps	#Holes	Result	Notes
		7	ON	50.02	147	441	4	6	correct	-
		8	ON	54.12	147	441	4	6	correct	-
25	Find the day of the week of a date string	1	ON	19.24	103	307	4	6	correct	-
		2	ON	19.22	103	307	4	6	correct	-
		3	ON	18.55	103	307	4	6	correct	-
		4	ON	18.98	103	307	4	6	correct	-
		5	ON	18.33	103	307	4	6	correct	-
		6	ON	18.11	103	307	4	6	correct	-
		7	ON	18.68	103	307	4	6	correct	-
		8	ON	18.16	103	307	4	6	correct	-
26	Compute age given date of birth	1	ON	11.82	142	288	3	4	correct	-
		2	ON	11.57	142	288	3	4	correct	-
		3	ON	11.71	142	288	3	4	correct	-
		4	ON	11.85	142	288	3	4	correct	-
		5	ON	11.9	142	288	3	4	correct	-
		6	ON	11.59	142	288	3	4	correct	-
		7	ON	11.67	142	288	3	4	correct	-
		8	ON	12.05	142	288	3	4	correct	-
27	Compute number of minutes between two time	1	OFF	-	-	-	-	-	incorrect	Fails to sythesize when pruning is on; OutOfMemoryError
		2	OFF	-	-	-	-	-	incorrect	Fails to sythesize when pruning is on; OutOfMemoryError
		3	OFF	-	-	-	-	-	incorrect	Fails to sythesize when pruning is on; OutOfMemoryError
		4	OFF	-	-	-	-	-	incorrect	Fails to sythesize when pruning is on; OutOfMemoryError
		5	OFF	-	-	-	-	-	incorrect	Fails to sythesize when pruning is on; OutOfMemoryError
		6	OFF	-	-	-	-	-	incorrect	Fails to sythesize when pruning is on; OutOfMemoryError
		7	OFF	-	-	-	-	-	incorrect	Fails to sythesize when pruning is on; OutOfMemoryError
		8	OFF	-	-	-	-	-	incorrect	Fails to sythesize when pruning is on; OutOfMemoryError
28	Compute number of seconds since the	1	ON	3.38	30	59	2	3	correct	-
		2	ON	3.3	30	59	2	3	correct	-

ID	Description	#Tests	Pruning	Synthesis Time (s)	#Paths	#Progs	#Comps	#Holes	Result	Notes
	midnight of a given time	3	ON	3.23	30	59	2	3	correct	-
		4	ON	3.28	30	59	2	3	correct	-
		5	ON	3.34	30	59	2	3	correct	-
		6	ON	3.21	30	59	2	3	correct	-
		7	ON	3.27	30	59	2	3	correct	-
		8	ON	3.31	30	59	2	3	correct	-
29	Compute the offset for a specified line in a document	1	ON	0.48	3	5	3	5	correct	-
		2	ON	0.5	3	5	3	5	correct	-
		3	ON	0.49	3	5	3	5	correct	-
		4	ON	0.49	3	5	3	5	correct	-
		5	ON	0.42	3	5	3	5	correct	-
		6	ON	0.41	3	5	3	5	correct	-
		7	ON	0.39	3	5	3	5	correct	-
		8	ON	0.47	3	5	3	5	correct	-
30	Get a paragraph element given its offset in the a document	1	ON	2.31	33	65	4	6	correct	-
		2	ON	3.77	33	65	4	6	correct	-
		3	ON	3.86	33	65	4	6	correct	-
		4	ON	3.46	33	65	4	6	correct	-
		5	ON	3.51	33	65	4	6	correct	-
		6	ON	2.27	33	65	4	6	correct	-
		7	ON	3.51	33	65	4	6	correct	-
		8	ON	3.44	33	65	4	6	correct	-
31	Obtain the title of a webpage specified by a URL	1	ON	68.89	289	577	3	4	correct	-
		2	ON	67.24	289	577	3	4	correct	-
		3	ON	51.82	289	577	3	4	correct	-
		4	ON	81.33	289	577	3	4	correct	-
		5	ON	53.9	289	577	3	4	correct	-
		6	ON	54.36	289	577	3	4	correct	-
		7	ON	52.88	289	577	3	4	correct	-
		8	ON	59.62	289	577	3	4	correct	-
32	Return doctype of XML document generated by string	1	ON	1.02	6	11	6	7	correct	-
		2	ON	0.87	6	11	6	7	correct	-
		3	ON	0.83	6	11	6	7	correct	-

ID	Description	#Tests	Pruning	Synthesis Time (s)	#Paths	#Progs	#Comps	#Holes	Result	Notes
		4	ON	0.79	6	11	6	7	correct	-
		5	ON	0.81	6	11	6	7	correct	-
		6	ON	0.79	6	11	6	7	correct	-
		7	ON	0.78	6	11	6	7	correct	-
		8	ON	0.92	6	11	6	7	correct	-
33	Generate an XML element from a string	1	ON	0.9	26	51	6	7	correct	-
		2	ON	0.88	26	51	6	7	correct	-
		3	ON	1.03	26	51	6	7	correct	-
		4	ON	0.95	26	51	6	7	correct	-
		5	ON	0.87	26	51	6	7	correct	-
		6	ON	0.97	26	51	6	7	correct	-
		7	ON	1.01	26	51	6	7	correct	-
		8	ON	0.95	26	51	6	7	correct	-
34	Read XML document from a file	1	ON	0.08	1	1	3	4	correct	-
		2	ON	0.31	1	1	3	4	correct	-
		3	ON	0.1	1	1	3	4	correct	-
		4	ON	0.11	1	1	3	4	correct	-
		5	ON	0.1	1	1	3	4	correct	-
		6	ON	0.11	1	1	3	4	correct	-
		7	ON	0.1	1	1	3	4	correct	-
		8	ON	0.1	1	1	3	4	correct	-
35	Generate an XML from file and query it using XPath	1	ON	36.67	24	52	7	10	correct	-
		2	ON	37.38	24	52	7	10	correct	-
		3	ON	37.84	24	52	7	10	correct	-
		4	ON	37.35	24	52	7	10	correct	-
		5	ON	37	24	52	7	10	correct	-
		6	ON	37.63	24	52	7	10	correct	-
		7	ON	36.84	24	52	7	10	correct	-
		8	ON	37.25	24	52	7	10	correct	-
36	Read XML document from a file and get the value of root attribute specified by a string	1	ON	0.36	3	5	5	7	correct	-
		2	ON	0.4	3	5	5	7	correct	-
		3	ON	0.34	3	5	5	7	correct	-
		4	ON	0.34	3	5	5	7	correct	-

ID	Description	#Tests	Pruning	Synthesis Time (s)	#Paths	#Progs	#Comps	#Holes	Result	Notes
		5	ON	0.35	3	5	5	7	correct	-
		6	ON	0.34	3	5	5	7	correct	-
		7	ON	0.37	3	5	5	7	correct	-
		8	ON	0.35	3	5	5	7	correct	-
37	Get number of children of root elements from xml string	1	ON	0.3	3	5	3	4	incorrect	-
		2	OFF	-	-	-	-	-	incorrect	Fails to sythesize when pruning is on; OutOfMemoryError
		3	OFF	-	-	-	-	-	incorrect	Fails to sythesize when pruning is on; OutOfMemoryError
		4	OFF	-	-	-	-	-	incorrect	Fails to sythesize when pruning is on; OutOfMemoryError
		5	OFF	-	-	-	-	-	incorrect	Fails to sythesize when pruning is on; OutOfMemoryError
		6	OFF	-	-	-	-	-	incorrect	Fails to sythesize when pruning is on; OutOfMemoryError
		7	OFF	-	-	-	-	-	incorrect	Fails to sythesize when pruning is on; OutOfMemoryError
		8	OFF	-	-	-	-	-	incorrect	Fails to sythesize when pruning is on; OutOfMemoryError
38	Get the version of xml string	1	ON	75.83	1356	2858	6	7	correct	-
		2	ON	77.97	1356	2858	6	7	correct	-
		3	ON	77.12	1356	2858	6	7	correct	-
		4	ON	78.79	1356	2858	6	7	correct	-
		5	ON	77.11	1356	2858	6	7	correct	-
		6	ON	79.36	1356	2858	6	7	correct	-
		7	ON	79.91	1356	2858	6	7	correct	-
		8	ON	76.11	1356	2858	6	7	correct	-
39	Calculate absolute value of an integer	1	OFF	0.05	1	1	1	2	correct	Fails to sythesize when pruning is on;
		2	OFF	0.05	1	1	1	2	correct	Fails to sythesize when pruning is on;
		3	OFF	0.06	1	1	1	2	correct	Fails to sythesize when pruning is on;
		4	OFF	0.05	1	1	1	2	correct	Fails to sythesize when pruning is on;
		5	OFF	0.05	1	1	1	2	correct	Fails to sythesize when pruning is on;
		6	OFF	0.05	1	1	1	2	correct	Fails to sythesize when pruning is on;
		7	OFF	0.06	1	1	1	2	correct	Fails to sythesize when pruning is on;
		8	OFF	0.05	1	1	1	2	correct	Fails to sythesize when pruning is on;

ID	Description	#Tests	Pruning	Synthesis Time (s)	#Paths	#Progs	#Comps	#Holes	Result	Notes
40	Increment an integer and return its old value	1	OFF	0.16	11	21	5	5	correct	Fails to sythesize when pruning is on;
		2	OFF	0.18	11	21	5	5	correct	Fails to sythesize when pruning is on;
		3	OFF	0.15	11	21	5	5	correct	Fails to sythesize when pruning is on;
		4	OFF	0.08	11	21	5	5	correct	Fails to sythesize when pruning is on;
		5	OFF	0.09	11	21	5	5	correct	Fails to sythesize when pruning is on;
		6	OFF	0.09	11	21	5	5	correct	Fails to sythesize when pruning is on;
		7	OFF	0.1	11	21	5	5	correct	Fails to sythesize when pruning is on;
		8	OFF	0.1	11	21	5	5	correct	Fails to sythesize when pruning is on;
41	Increment an integer by 2 and return its old value	1	OFF	-	792	24139	-	-	incorrect	Fails to sythesize when pruning is on; TIMEOUT
		2	OFF	-	824	27278	-	-	incorrect	Fails to sythesize when pruning is on; TIMEOUT
		3	OFF	-	864	30576	-	-	incorrect	Fails to sythesize when pruning is on; TIMEOUT
		4	OFF	-	919	33295	-	-	incorrect	Fails to sythesize when pruning is on; TIMEOUT
		5	OFF	-	865	30699	-	-	incorrect	Fails to sythesize when pruning is on; TIMEOUT
		6	OFF	-	821	27045	-	-	incorrect	Fails to sythesize when pruning is on; TIMEOUT
		7	OFF	-	920	33416	-	-	incorrect	Fails to sythesize when pruning is on; TIMEOUT
		8	OFF	-	929	34237	-	-	incorrect	Fails to sythesize when pruning is on; TIMEOUT
42	Get the class name of an object	1	OFF	0.43	8	15	2	3	correct	Fails to sythesize when pruning is on;
		2	OFF	0.39	8	15	2	3	correct	Fails to sythesize when pruning is on;
		3	OFF	0.45	8	15	2	3	correct	Fails to sythesize when pruning is on;
		4	OFF	0.34	8	15	2	3	correct	Fails to sythesize when pruning is on;
		5	OFF	0.47	8	15	2	3	correct	Fails to sythesize when pruning is on;
		6	OFF	0.35	8	15	2	3	correct	Fails to sythesize when pruning is on;
		7	OFF	0.39	8	15	2	3	correct	Fails to sythesize when pruning is on;
		8	OFF	0.35	8	15	2	3	correct	Fails to sythesize when pruning is on;
43	Get the first value of an integer array	1	ON	0.04	3	5	4	4	incorrect	-
		2	ON	0.03	3	5	4	4	incorrect	-
		3	ON	0.05	3	5	4	4	incorrect	-

ID	Description	#Tests	Pruning	Synthesis Time (s)	#Paths	#Progs	#Comps	#Holes	Result	Notes
		4	ON	0.04	3	5	4	4	incorrect	-
		5	OFF	0.05	2	3	2	3	correct	Fails to sythesize when pruning is on;
		6	OFF	0.04	2	3	2	3	correct	Fails to sythesize when pruning is on;
		7	OFF	0.03	2	3	2	3	correct	Fails to sythesize when pruning is on;
		8	OFF	0.03	2	3	2	3	correct	Fails to sythesize when pruning is on;
44	Calculate minimum value between two integers	1	OFF	0.05	1	1	1	3	correct	Fails to sythesize when pruning is on;
		2	OFF	0.05	1	1	1	3	correct	Fails to sythesize when pruning is on;
		3	OFF	0.05	1	1	1	3	correct	Fails to sythesize when pruning is on;
		4	OFF	0.05	1	1	1	3	correct	Fails to sythesize when pruning is on;
		5	OFF	0.05	1	1	1	3	correct	Fails to sythesize when pruning is on;
		6	OFF	0.05	1	1	1	3	correct	Fails to sythesize when pruning is on;
		7	OFF	0.05	1	1	1	3	correct	Fails to sythesize when pruning is on;
		8	OFF	0.05	1	1	1	3	correct	Fails to sythesize when pruning is on;
45	Calculate minimum value between three integers	1	ON	0.05	1	1	2	4	incorrect	-
		2	OFF	0.22	6	66	2	5	correct	Fails to sythesize when pruning is on;
		3	OFF	0.42	6	66	2	5	correct	Fails to sythesize when pruning is on;
		4	OFF	0.22	6	66	2	5	correct	Fails to sythesize when pruning is on;
		5	OFF	0.24	6	66	2	5	correct	Fails to sythesize when pruning is on;
		6	OFF	0.23	6	66	2	5	correct	Fails to sythesize when pruning is on;
		7	OFF	0.22	6	66	2	5	correct	Fails to sythesize when pruning is on;
		8	OFF	0.22	6	66	2	5	correct	Fails to sythesize when pruning is on;
46	Given an array, set the last entry the value of first entry	1	ON	0.14	7	13	5	5	incorrect	-
		2	OFF	2.61	314	2032	7	7	correct	Fails to sythesize when pruning is on;
		3	OFF	2.29	314	2032	7	7	correct	Fails to sythesize when pruning is on;
		4	OFF	2.92	314	2032	7	7	correct	Fails to sythesize when pruning is on;
		5	OFF	2.51	314	2032	7	7	correct	Fails to sythesize when pruning is on;
		6	OFF	2.77	314	2032	7	7	correct	Fails to sythesize when pruning is on;
		7	OFF	2.5	314	2032	7	7	correct	Fails to sythesize when pruning is on;
		8	OFF	2.48	314	2032	7	7	correct	Fails to sythesize when pruning is on;
47	Sort an integer array	1	ON	0.27	1	1	2	2	correct	-
		2	ON	0.11	1	1	2	2	correct	-

ID	Description	#Tests	Pruning	Synthesis Time (s)	#Paths	#Progs	#Comps	#Holes	Result	Notes
		3	ON	0.12	1	1	2	2	correct	-
		4	ON	0.11	1	1	2	2	correct	-
		5	ON	0.11	1	1	2	2	correct	-
		6	ON	0.11	1	1	2	2	correct	-
		7	ON	0.11	1	1	2	2	correct	-
		8	ON	0.12	1	1	2	2	correct	-

References

[1] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. 2011. Synthesis of loop-free programs. In *PLDI 2011*. 62–73.

[2] S. Jha, S. Gulwani, S. Seshia, and A. Tiwari. 2010. Oracle-guided component-based program synthesis. In *ICSE 2010*. 215–224.

[3] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-Based Synthesis for Complex APIs. In *POPL 2017*. 599–612.

[4] Armando Solar-Lezama. 2013. Program sketching. *STTT* 15, 5-6 (2013), 475–495.

[5] Shaon Barman, Rastislav Bodik, Satish Chandra, Joel Galenson, Doug Kimelman, Casey Rodarmor, and Nicholas Tung. 2010. Programming with angelic nondeterminism. In *POPL 2010*. 339–352.

[6] Jinru Hua and Sarfraz Khurshid. 2017. Sketch4J: Execution-Driven Sketching for Java. In *SPIN 2017*. To appear.

[7] Patrice Godefroid. 1997. Model Checking for Programming Languages using Verisoft. In *POPL 1997*. 174–186.

[8] V. Debroy and W.E. Wong. 2010. Using mutation to automatically suggest fixes for faulty programs. In *ICST 2010*. 65–74.

[9] H.D.T. Nguyen, D. Qi, A. Roychoudhury and S. Chandra. 2013. SemFix: program repair via semantic analysis. In *ICSE 2013*. 772–781.

[10] Jinru Hua and Sarfraz Khurshid. 2016. A Sketching-Based Approach for Debugging Using Test Cases. In *ATVA 2016*. 463-478.